1. 上课约定须知

课程主题：MapReduce 分布式计算应用程序编程框架--第二次课（源码剖析）
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次作业复盘

需求：数字排序并加序号

源数据理解：

第一个数据文件：data-source-1.data

```
2
32
654
32
15
```

第二个数据文件：data-source-2.data

```
756
65223
5956
22
650
92
26
54
6
```

最终结果：

第一个结果文件：result-0

```
1  2
2  6
3  15
4  22
5  26
```

第二个结果文件：result-1

```
6  32
7  32
8  54
9  92
10 650
```

第三个结果文件：result-2

```
11 654
12 756
13 5956
14 65223
```

作业要求

```
1、不能在本地运行得到结果，必须在集群运行得到结果
2、必须不能使用一个 ReduceTask 去执行，必须使用多个 ReduceTask 来执行。
```

3. 上次内容总结

主要讲解的是 MapReduce 的应用实践 和 执行原理相关：

```
1、MapReduce的架构设计和工作原理
    MapReduce架构设计
    MapReduce的核心Shuffle流程
    MapReduce并行度决定机制
2、MapReduce企业最佳实践
    MapReduce程序编写规范总结
    MapReduce序列化和分区分组
    MapReduce Join实现
```

学好 MapReduce，总结一起，就是三个方面：

- 一个编程作业题

  ```
  针对海量数据排序，并且加序号
  ```

- 一个核心Shuffle流程

  ```
  Shuffle是分布式引擎必然要支持的一个核心工作流程，了解是理解分布式计算引擎的最核心问题
  ```

- 两个原理问题

  ```
  1、如果一个MR在执行过程中，有reducer阶段，那么就一定会排序？为什么一定要排序呢？
  2、为什么，MapReduce的数据处理模型，被设计成是key-value形式的？
  ```

4. 本次内容大纲

本次课程主要讲解的是 MapReduce 的源码分析，分为以下几个方面：

```
1、MapReduce WordCount 源码解读
2、MapReduce 任务提交脚本分析
3、MapReduce 的任务提交源码详解
4、MapReduce 逻辑切片
5、ResourceManager 处理任务提交
6、AppMaster 启动和 Task 启动
7、MapTask 执行源码详解
8、RecordReader 读取数据源码剖析
9、ReduceTask 执行源码详解
10、RecordWriter 写出数据源码剖析
11、MapReduce 的Shuffle源码详解
```

5. MapReduce 完整执行流程源码解析

5.1. WordCount源码解读

看 WordCount 的源码编写，更多的是去总结一个 MR 的编写套路

在 Hadoop-2.7.7 这个版本的源码项目中， WordCount 的例子程序，位于 hadoop-mapreduce-project 子项目中的 hadoop-mapreduce-examples 子项目中。关于 WordCount 的详细解释，标注在源码中。

5.2. MapReduce任务提交脚本分析

当一个 MapReduce 被编写完成需要执行的时候，是这样的方式：

```
1、先编写 MapReduce 应用程序
2、打成 jar 包
3、通过 hadoop jar 的命令形式来提交这个 jar  运行
    hadoop jar examples.jar wordcount /wc/input/ /wc/output/
    hadoop 这shell脚本：如果参数是jar, class=RunJar
4、最终跳转到： RunJar 这个类来运行。
    RunJar.main()
5、通过反射的方式最终跳转到用户 主类的 main 方法运行
    Driver.main()
    这里面是包含：job.submit()
```

5.3. MapReduce 的任务提交源码详解

此时，开始执行用户的主类的 main() 方法，执行 job.waitForCompetetion() 来等待任务执行完成。

在 Job.submit() 方法中，最重要的是：

```
// 设置mapreduce的状态为： DEFINE
ensureState(JobState.DEFINE);

// 设置启用new API
setUseNewAPI();

// 获取提交客户端，链接YARN集群
connect();

// 获取提交器
submitter = getJobSubmitter(cluster.getFileSystem(), cluster.getClient());
    submitter.submitJobInternal(Job.this, cluster);

// 提交之后，设置 mapreduce 的状态为： RUNNING
state = JobState.RUNNING;
```

connect() 方法做的主要的事情：

```
1、Job 的内部有一个 Cluster cluster 成员变量
2、Cluster 的内部有一个 YARNRunner client 的成员变量
3、YARNRunner 的内部有一个 ResourceMgrDelegate resMgrDelegate 成员变量
4、ResourceMgrDelegate 的内部有一个 YarnClientImpl client 成员变量
5、YarnClientImpl 的内部有一个 ApplicationClientProtocol rmClient 的成员变量
```

最终：rmClient 这个东西，就是 RM 的代理对象！

```
最终提交任务是通过：rmClient.submitApplication();
```

接下来再看提交过程：

```
# 第一步：通过 JobSubmitter 提交 Job
# 在这一步，会做很多一些细节操作。
JobSubmitter.submitJobInternal(Job.this, cluster);
// 当进入 JobSubmitter 的提交步骤的时候，内部做了很多细节操作
// 生成了 JobID
// 重要的事情：包括：1、Task启动所需要的资源生成好并且防止在HDFS上， 2、执行逻辑切片

# 第二步：通过 YARNRunner 来提交 Job
submitClient.submitJob(jobId, submitJobDir.toString(), job.getCredentials());

# 第三步：通过 ResourceMgrDelegate 来提交 Job，获取了 ApplicationID
resMgrDelegate.submitApplication(appContext);

# 第四步：通过 YarnClientImpl 提交 ApplicationID
client.submitApplication(appContext);

# 第五步：通过 ResrouceManager 的代理对象：ApplicationClientProtocol 来提交
ApplicationID
rmClient.submitApplication(request);
```

最终：ResourceManager 组件中的 ClientRMService 来执行 submitApplication() 的 RPC 服务处理

5.4. MapReduce 逻辑切片

入口方法：JobSubmiter.writeSplits(job, submitJobDir);

内部真正执行切片的是：FileInputFormat.getSplits(job);

核心逻辑为：

```
// 存储逻辑切片结果
List<InputSplit> splits = new ArrayList<InputSplit>();

// 遍历每个文件
for(FileStatus file : files) {

    long length = file.getLen();
    if(length != 0) {
        // 如果文件可切分，则执行逻辑切片
        if(isSplitable(job, path)) {
            // 计算逻辑切片大小
            long splitSize = computeSplitSize(blockSize, minSize, maxSize);

            // 只要剩下的该文件的大小大于 splitSize 的 1.1 倍 就执行逻辑切片
            while(((double) bytesRemaining) / splitSize > SPLIT_SLOP) {
                // 切片
                splits.add(makeSplit(file, start, length, hosts, inMemoryHosts);
                // 管理文件剩余大小
                bytesRemaining -= splitSize;
            }
```

```
            // 如果剩余的数据大小 不等于0， 并且小于 splitSize 的 1.1 倍，则划分成一个逻辑切片
            if(bytesRemaining != 0) {
                splits.add(makeSplit(file, start, length, hosts, inMemoryHosts);
            }
        // 否则直接形成一个逻辑切片
        }else{
            splits.add(makeSplit(file, start, length, hosts, inMemoryHosts);
        }
    }else{
        // 什么也没做
    }
}

return splits;
```

两个问题：

```
1、200个200M的文件，最终启动多少个MapTask? 400
2、200个130M的文件，最终启动多少个MapTask? 200
2、200个140M的文件，最终启动多少个MapTask? 200      （128 + 12.8）
2、200个145M的文件，最终启动多少个MapTask? 400      （128 + 12.8）
```

5.5. ResourceManager 处理任务提交

如果 client 提交 MR 应用程序给 YARN，最终是 RM 接收到之后，由它的成员变量： ClientRMService 来执行处理

核心入口方法：ClientRMService.submitApplication(SubmitApplicationRequest request)

内部核心逻辑：

```
// 第一步：通过 RMAppManager 提交 Application
rmAppManager.submitApplication(submissionContext, System.currentTimeMillis(), user);

// 第二步：提交 RMAppEvent(applicationId, RMAppEventType.START) 事件给 Dispatcher
// 内部将 RMAppEvent 加入到：AsyncDispatcher 的 eventQueue 队列中
// 内部会调用 Dispatcher 的 Handler 执行事件处理
this.rmContext.getDispatcher().getEventHandler().handle(new RMAppEvent(applicationId, RMAppEventType.START));
GenericEventHandler.run(){
    eventQueue.put(event);
}

// 第三步：调度 RMAppEvent，在初始化 AsyncDispatcher 的时候，会
createThread(){
    event = eventQueue.take();
    dispatch(event);
}.start()

// 第四步：调用 Handler 执行：Event 的处理
dispatch(Event event){
```

```
    // RMAppEventType = ApplicationEventDispatcher
    EventHandler handler = eventDispatchers.get(type);
    handler.handle(event);
}

// 第五步：调用 RMAppImpl 的 handle方法处理 Event
rmApp.handle(event);

// 第六步：调用 StateMachineFactory 的 doTransition(event.getType(), event); 执行处理
stateMachine.doTransition(event.getType(), event);
```

之后，就是初始化和注册 Application，向 ResourceManager 申请 Container 启动 MRAppMaster，然后 MRAppMaster 解析任务启动 MapTask 和 ReduceTask，这都属于 YARN 的调度源码，再次不多讲了。

```
1、向 ApplicatoinMaster 注册 Application
2、向 ResourceManager 申请一个 Container来启动 MR Application 的主控程序：MRAppMaster
3、MRAppMaster 向 ResourceManager 申请 container 来启动 MapTask
4、等待 MapTask 执行完毕，再启动 ReduceTask
```

事实上，ReduceTask 并不需要等待所有的 MapTask 执行完毕，当 大部分的 MapTask 执行完毕的时候，就会启动 ReduceTask 来提前拉取执行完毕的 MapTask 的结果数据，提前做合并

当 MapTask 执行完毕了之后，会跟 MRAppMaster 执行汇报，所以，每个启动的 ReduceTask 就能够知道，那些节点上的那些 MapTask 执行完毕了，可以拉取数据了。

5.6. AppMaster 启动和 Task 启动

启动两类程序：

```
1、主控程序 MRAppMaster ： 用来监控，调度，协调，容错，推测执行 Task 等
2、Task程序 YarnChild ： 真正干活的程序
```

AppMaster 核心入口方法：MRAppMaster.main() 方法

MapTask 启动的入口：YarnChild.main()

ReduceTask 启动的入口：YarnChild.main()

5.7. MapTask 执行源码详解

MapTask 启动的入口：YarnChild.main()

MapTask.run(final JobConf job, final TaskUmbilicalProtocol umbilical)

核心逻辑：

```
YarnChild.main()

    MapTask.run(final JobConf job, final TaskUmbilicalProtocol umbilical)

        # run 方法中，前提做一些准备，最后调用 runNewMapper 来执行一个 MapTask 中的业务逻辑处理
        runNewMapper(job, splitMetaInfo, umbilical, reporter);

            # 创建一个 Mapper 实例对象
            ReflectionUtils.newInstance(taskContext.getMapperClass(), job);

            # 调用 Mapper 组价的 run() 来执行业务处理
            mapper.run(mapperContext);
```

注意 mapper.run(mapperContext); 的参数 mapperContext，内部包装了 mapContext， 他是 MapContextImpl 的实例对象，内部包含了 input，output 等组件。这是最重要的封装。请看 Mapper.run() 方法的实现就可了解：

```
public void run(Context context) throws IOException, InterruptedException {
    // 生命周期方法
    setup(context);
    try {
        while (context.nextKeyValue()) {
            // 业务处理方法
            map(context.getCurrentKey(), context.getCurrentValue(), context);
        }
    } finally {
        // 生命周期方法
        cleanup(context);
    }
}
```

其中，需要注意 context 的四大方法：

```
context.nextKeyValue()
context.getCurrentKey()
context.getCurrentValue()
context.write(outkey, outvalue)
```

其实：

1、前三个就是调用：MapContextImpl 内部的 input（RecordReader） 对象的同名方法。

2、后一个方法，就是调用 MapContextImpl 内部的 output（NewOutputCollector）执行 Mapper阶段的数据写出

5.8. RecordReader 读取数据源码剖析

RecordReader 是 MapReduce 中，真正负责读取数据的组件。具体怎么读取数据，由 InputFormat 来决定，因为 RecordReader 就是 InputFormat 来负责创建的。在 MapReduce 中的，这两个组件的默认实现是：

```
InputFormat: TextInputFormat
RecordReader：LineRecordReader
```

默认的读取数据的核心方法的入口：LineRecordReader.nextKeyValue()

```
public boolean nextKeyValue() throws IOException {

    if (key == null) {
        key = new LongWritable();
    }
    key.set(pos);

    if (value == null) {
        value = new Text();
    }

    int newSize = 0;
    while (getFilePosition() <= end || in.needAdditionalRecordAfterSplit()) {
        if (pos == 0) {
            newSize = skipUtfByteOrderMark();
        } else {
            // 真正读取数据
            // in = LineReader
            newSize = in.readLine(value, maxLineLength, maxBytesToConsume(pos));
            pos += newSize;
        }

        if ((newSize == 0) || (newSize < maxLineLength)) {
            break;
        }

        // line too long. try again
        LOG.info("Skipped line of size " + newSize + " at pos " + (pos -
newSize));
    }

    if (newSize == 0) {
        key = null;
        value = null;
        return false;
    } else {
        return true;
    }
}
```

关于断行的处理，可以详细理解：newSize = in.readLine(value, maxLineLength, maxBytesToConsume(pos)); 这句代码的详细实现。具体处理：

```
1、第一个 InputSplit，读取数据到下一个 InputSplit 的第一个行分隔符处
2、从第二个 InputSplit 开始的每一个InputSplit都要从第一个行分隔符处开始读取，因为前半行断行，已经被上一个 InputSplit 读取过了。
3、最后一个 InputSplit 读取到自然结束即可
```

5.9. ReduceTask 执行源码详解

启动入口：YarnChild.main()

ReduceTask 启动的入口：ReduceTask.run(final JobConf job, final TaskUmbilicalProtocol umbilical)

核心逻辑：

```
YarnChild.main()

    ReduceTask.run(final JobConf job, final TaskUmbilicalProtocol umbilical)

        # run 方法中，前提做一些准备，最后调用 runNewReducer 来执行一个 ReduceTask 中的
业务逻辑处理
        runNewReducer(job, umbilical, reporter, rIter, comparator, keyClass,
valueClass);

            # 创建 Reducer 实例对象


            # 调用 Reducer 组价的 run() 来执行业务处理
            reducer.run(reducerContext);
```

注意 reducer.run(reducerContext); 的参数 mapperContext，内部包装了 reduceContext， 他是 ReduceContextImpl 的实例对象，内部包含了 output 等组件。这是最重要的封装。请看 Reducer.run() 方法的实现就可了解：

```
public void run(Context context) throws IOException, InterruptedException {
    // 调用一次 setup方法
    setup(context);
    try {
        // 循环读取数据（两个判断条件：是否有下一个record， 是否是同一个key的record）
        while (context.nextKey()) {

            // 调用reduce方法 执行的数据是 key 相同的 一组 key-value values 的值
            // context.getValues()
            reduce(context.getCurrentKey(), context.getValues(), context);

            // If a back up store is used, reset it
            Iterator<VALUEIN> iter = context.getValues().iterator();
            if (iter instanceof ReduceContext.ValueIterator) {
                ((ReduceContext.ValueIterator<VALUEIN>)
iter).resetBackupStore();
            }
        }
    } finally {

        // 最后调用 cleanup 进行收尾
        cleanup(context);
    }
}
```

其中，需要注意 context 的四大方法：

```
context.nextKey()
context.getCurrentKey()
context.getValues()
context.write(outkey, outvalue)
```

关于 key 和 values 的理解： 理论上来说，这些 values 是属于一个相同的key 的，但是这些key 只是在判断逻辑处理中返回了 0 而已，并不表示，这变量完全一样！当你在迭代 values 的时候，key 也在相应的变化！

先看第一个方法：nextKey()

```
public boolean nextKey() throws IOException, InterruptedException {
    while(hasMore && nextKeyIsSame) {
        // 在这个方法里 hasMore 状态会改变
        nextKeyValue();
    }

    // 然后再继续判断，看是否有下一个key， 有的话，key计数器加1， 然后读取一个出来
    if(hasMore) {
        if(inputKeyCounter != null) {
            inputKeyCounter.increment(1);
        }
        return nextKeyValue();
    } else {
        return false;
    }
}
```

非常经典的一个控制逻辑，它的核心逻辑是：

```
1、方法的返回值是：代表有无读取到一组数据的布尔值
2、nextKey() 是读取一组数据，但是方法返回值是代表有无读取到数据的布尔值，那么读取到的数呢？
```

看看 nextKeyValue(); 的核心逻辑即可：

```
public boolean nextKeyValue() throws IOException, InterruptedException {
    if(!hasMore) {
        key = null;
        value = null;
        return false;
    }
    firstValue = !nextKeyIsSame;

    /************************************************
     * TODO_MA 马中华 https://blog.csdn.net/zhongqi2513
     *  注释： 取值 key
     */
    DataInputBuffer nextKey = input.getKey();
    currentRawKey.set(nextKey.getData(), nextKey.getPosition(),
nextKey.getLength() - nextKey.getPosition());
    buffer.reset(currentRawKey.getBytes(), 0, currentRawKey.getLength());

    // 反序列化key
    key = keyDeserializer.deserialize(key);
```

```
    /**************************************************
     * TODO_MA 马中华 https://blog.csdn.net/zhongqi2513
     *  注释： 取值 value
     */
    DataInputBuffer nextVal = input.getValue();
    buffer.reset(nextVal.getData(), nextVal.getPosition(), nextVal.getLength() -
nextVal.getPosition());

    // 反序列化 value
    value = valueDeserializer.deserialize(value);

    currentKeyLength = nextKey.getLength() - nextKey.getPosition();
    currentValueLength = nextVal.getLength() - nextVal.getPosition();

    if(isMarked) {

        /**************************************************
         * TODO_MA 马中华 https://blog.csdn.net/zhongqi2513
         *  注释： 暂时缓存当前 nextKeyValue方法读取到的一组  key 相同的 key-value
         */
        backupStore.write(nextKey, nextVal);
    }

    /**************************************************
     * TODO_MA 马中华 https://blog.csdn.net/zhongqi2513
     *  注释： 读取下一个 key，如果能读取到， 则判断这个读取的key和上一个key是否是一致的。
     *  如果没有读取到， 则直接返回 false ， 表示没有读取到下一个， 并且nextKeyIsSame值也
是false
     */
    hasMore = input.next();
    if(hasMore) {
        // TODO_MA 注释：读取一个key
        nextKey = input.getKey();
        // TODO_MA 注释：判断是否跟上一个key一致
        nextKeyIsSame = comparator.compare(currentRawKey.getBytes(), 0,
                    currentRawKey.getLength(), nextKey.getData(),
nextKey.getPosition(),
                    nextKey.getLength() - nextKey.getPosition()) == 0;
    } else {
        nextKeyIsSame = false;
    }

    // TODO_MA 注释：输入 value 个数 的计数器 + 1
    inputValueCounter.increment(1);
    return true;
}
```

所以，其实：

```
1、context.getKey() 读取到的就是 ReduceContextImpl 中的 key 成员变量
2、context.getValues() 读取到的就是 ReduceContextImpl 中的 backupStore 成员变量中的
values
```

5.10. RecordWriter 写出数据源码剖析

RecordWriter 是 MapReduce 中，真正负责写出数据的组件。具体怎么写出数据，由 OutputFormat 来决定，因为 RecordWriter 就是 OutputFormat 来负责创建的。在 MapReduce 中的，这两个组件的默认实现是：

```
OutputFormat: TextOutputFormat
RecordWriter：LineRecordWriter
```

默认的写出数据的核心方法的入口：LineRecordWriter.write()

```
public synchronized void write(K key, V value) throws IOException {

    // TODO_MA 注释： 判断 key 是否为空
    boolean nullKey = key == null || key instanceof NullWritable;

    // TODO_MA 注释： 判断 value 是否为空
    boolean nullValue = value == null || value instanceof NullWritable;

    // TODO_MA 注释： 如果 key value 都为空，则直接返回
    if (nullKey && nullValue) {
        return;
    }

    // TODO_MA 注释：写出 key
    if (!nullKey) {
        writeObject(key);
    }

    // TODO_MA 注释： 如果 key 和 value 都不为空，则写出 分隔符，默认是 \t
    if (!(nullKey || nullValue)) {
        out.write(keyValueSeparator);
    }

    // TODO_MA 注释： 输出 value
    if (!nullValue) {
        writeObject(value);
    }

    // TODO_MA 注释： 写出换行符： \n
    out.write(newline);
}
```

总结一下：

```
1、如果 key 不为空，则写出 key
2、如果 key 和 value 都不为空，则写出 分隔符，默认是 \t
3、如果 value 不为空，输出 value
4、最后，写出换行符：\n
```

5.7 - 5.10 这四个知识点的总结一下：

```
1、MapTask 的输入数据是怎么被读取进来的
2、ReduceTask 的输入一组数据是怎么被读取到的
3、ReduceTask 的计算结果数据是怎么写出到磁盘文件的
```

缺一个：

```
MapTask 的数据是怎么写出到 ReduceTask 的?
```

入口：MapTask 的 map() 方法中的 context.write(key, value)

5.11. MapReduce 的Shuffle 详解

在 NewOutputCollector 内部有一个成员变量：MapOutputCollector collector，具体实现是：MapOutputBuffer， 当 Mapper 组件，处理完数据，执行输出的时候，是这样的逻辑：

```
protected void map(KEYIN key, VALUEIN value, Context context) throws
IOException, InterruptedException {
    context.write((KEYOUT) key, (VALUEOUT) value);
}
```

内部调用：

```
collector.collect(key, value, partitioner.getPartition(key, value, partitions));
```

这个时候，Shuffle 开始!

5.11.1. Partitioiner

Shuffle 过程中，第一个参与工作的就是：Partitioner，默认实现是： HashPartitioiner，看具体实现：

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {

    // numReduceTasks =  分区个数  =  reduceTask 个数

    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

5.11.2. MapOutputBuffer

当 Mapper 阶段输出数据的时候，首先被写出到：MapOutputBuffer 这个环形缓冲区（首尾相连的连续内存块，字节数组实现），看 collect() 方法的具体实现：

```
1、为写入数据做准备
2、判断是否有足够空间去写入，如果不够，执行溢写，或者等待溢写完成，
3、溢写完成之后，执行真正写入
4、写入数据和元数据
```

如果要执行溢写：startSpill() 方法是入口，负责执行溢写的是一个叫做 SpillThread 的线程。

5.11.3. SpillThread

SpillThread 是专门用来执行溢写动作的一个线程，在 MapOutputBuffer 初始化的时候，就已经启动了。启动好了之后，随时等待：spillReady.signal();

其实，在 MapTask 工作的时候，有两个重要的线程：

```
1、执行 context.write(key,value) 的线程，叫做写数据线程
2、执行 spill 的 SpillThread 线程， 叫做溢写线程
```

当 MapOutputBuffer 的 kvBuffer 装满 80% 的时候，就要锁定这 80% 的数据，获取一个 Lock

当锁定了也就表示，写数据线程 会给 溢写线程 发送一个 spillReady.signal()

```
1、spillDone 用来表示 spill 完成
2、spillReady 用来表示可以执行 spill
```

看具体逻辑：

```
1、当初始化 MapOutputBuffer 的时候，就启动了 SpillThread，一上来，SpillThread 就阻塞在：spillReady.await(); 等待执行 spill 的信号
2、当写数据线程 写入数据到 80% 的时候， 发送 spillReady.signal()
3、当 接收到 spillReady.signal() 的时候。 spillThread 就执行这 80% 数据的溢写。当然：溢写的时候会县排序，然后执行 Conbiner
4、当溢写完成，就发送： spillDone.signal()
5、如果原来因为资源不足（剩下的20%的空间，也被写数据线程写满了）写数据线程就会阻塞在：spillDone.await()
```

为了理解 kvbuffer 和 SpillThread 的交互，还需要理解两个重要的信号变量 和一个锁变量：

```
1、spillReady
    当需要执行 spill 的时候，写数据线程执行 spillReady.signal(); 发送信号给溢写数据线程

2、spillDone
    当 spill 完成的时候，溢写数据执行：spillDone.signal(); 发送信号给写数据线程

3、spillLock
    当需要进行 spill 的时候，就锁住这 80% 的数据
```

真正执行溢写的方法是：sortAndSpill();，请看 SpillThread 线程的 run() 方法的具体实现：

```
protected class SpillThread extends Thread {
    @Override
    public void run() {
        spillDone.signal();

        while(!spillInProgress) {
            // TODO_MA 注释：等待 sortAndSpill 的信号
            spillReady.await();
        }

        // 执行溢写
        sortAndSpill();
    }
}
```

5.11.4. Sorter

在 Shuffle 过程中，如果要执行溢写了，则溢写最开始要做的动作就是：

```
sorter.sort(MapOutputBuffer.this, mstart, mend, reporter);
```

给待溢写的数据排序！当排序之后，执行分区分别写出到磁盘文件，并生成索引数据文件。

5.11.5. Combiner

在排序之后，执行分区溢写的时候，还会执行 combine 动作：

```
combinerRunner.combine(kvIter, combineCollector);
```

6. 本次课程总结

本次课程的主要内容： MapReduce 的完整执行流程的源码详解，包含的主要内容项：

```
1、MapReduce WordCount 码解读
2、MapReduce 任务提交脚本分析
3、MapReduce 的任务提交源码详解
4、MapReduce 逻辑切片
5、ResourceManager 处理任务提交
6、AppMaster 启动和 Task 启动
7、MapTask 执行源码详解
8、RecordReader 读取数据源码剖析
9、ReduceTask 执行源码详解
10、RecordWriter 写出数据源码剖析
11、MapReduce 的Shuffle源码详解
```

通过这个完整流程的源码阅读，了解到最开始产生的分布式计算引擎的执行原理。以及详细的执行细节。这对于我们学习其他的分布式组件是非常有用的！

7. 本次课程作业

同上次作业布置！

具体作业：对海量分布式数据执行排序之后，加全局序号